Object-Oriented Programming in Java

Problem Set 1 Due: Jan 8, 2001

Exercises

Exercise 1: Hello World

Compile and run the Hello.java program. Modify the initial message, recompilem, and re-run. Add to Hello the recursive and iterative factorial methods from class. Test these with calls from main().

Exercise 2: Arrays

Write and test static methods

```
float Mean(float[] data)
float Variance(float[] data)
```

which compute the mean and variance of the elements in data respectively.

Exercise 3: Recursion and Iteration

Write a method that computes fib(x) recursively using the recurrence

$$\begin{aligned} fib(0) &= 1 \\ fib(1) &= 1 \\ fib(x) &= fib(x-1) + fib(x-2). \end{aligned}$$

Now write a method that computes fib(x) iteratively. (Hint: Start from 0, and compute the sum forward, saving the last two values each iteration).

Test these by calling from main()

Exercise 4: Command Arguments

Modify your main() routine from Exercise 3 to take two arguments on the command line. The first is a string with values “I” or “R” which selects the Interative or Factorial versions respectively. The second is an integer which is the arg to $fib()$ (Remember to convert this from String to int. Hint: Use Integer.parseInt()).

Use the UNIX `time` command to time both versions on $fib(10)$ and $fib(40)$. Is there a moral here?

Exercise 5: Arrays: A Word Problem

If one lends D dollars at an (annual compound) interest rate of r (ie r = 0.07 for 7% interest). The investment, at the end of N years, will be worth $D(1+r)^N$ dollars. Inverting this, to have a nest-egg of D dollars in N years, one must invest $D/(1+r)^N$ dollars today. This quantity is called the Net Present Value or Discounted Value of D dollars N years in the future. Some investments, like stocks and bonds, produce an annual cash flow (D_t for year t). The net present value of such an asset is computed by summing the Discounted cash flows for each year.

$$NPV = \sum_t (D_t/(1+r)^t)$$

a) Write a static method to compute the NPV of an investment returning a constant $D_t = d$ dollars for N years assuming an interest rate of r. (Assume the first payment comes in year 0 and is not discounted).

b) If you win $2,000,000 in the Lottery, rather than a check for $2M, you will actually receive $100,000 per year for the next 20 years. Call your method to calculate the NPV of your prize assuming r=0.06 (ie 6%). Compute the NPV for a 9% interest rate.

Problems

In this section, we will implement a number of simple numerical algorithms and data structures in Java. [1]

Problem 1: Root Finding by Binary Search

One algorithm for computing (approximately) the real roots of a continuous function (i.e.; a value R such that $f(R) = 0$), is binary search. Suppose you know two values a and b such that $f(a)$ and $f(b)$ have opposite signs. By the intermediate value theorem, there is at least one root of f between a and b. The root is said to be *bracketed* by a and b.

We can now trap the root to any accuracy by computing $f(x)$, where $x = (a+b)/2$ and comparing to $f(a)$ and $f(b)$. If $f(x)$ has the same sign as $f(a)$, we know $x < R < b$, and if $f(x)$ has the same sign as $f(b)$, we know $a < R < x$. We can iterate this procedure to compute the root to any desired accuracy. Note that the error in our estimate is halved each iteration.

Write a driver class `FunctionTest` and implement the root bracketing algorithm as a static method:

```
public static double bracketRoot(double a, double b, double maxerr);
```

Use `Math.sin()` as the function to be evaluated. Call this method from `main()` to find the root of $sin(x)$ between 3 and 4 to within 10^-8.

Save your code. You will submit it as part of Problem 4.

Problem 2: Numerical Integration: Extended Trapezoidal Rule

Another useful numerical algorithm computes definite integrals through use of the trapezoidal approximation

$$\int_a^b f(x)dx \approx (b-a)\frac{f(a)+f(b)}{2}.$$

[1] This is not to imply that Java is the language of choice for numerical algorithms, or algorithms in general. Algorithms are generally about program speed and efficiency, Java and OOP are about *programmer* speed and efficiency in constructing large systems. To quote *Numerical Recipes in C*, "The practical scientist is trying to solve tomorrow's problems with today's hardware, computer scientists, it often seems, are doing the reverse."

This (crude) approximation can be made more accurate by dividing the interval $[a, b]$ into N segments of length $h = (b - a)/N$, computing the integral over each of the segments and summing the result. [2] The resulting approximation is

$$\sum_{i=0}^{N-1} h \frac{f(a+hi) + f(a+h(i+1))}{2}.$$

Since the each intermediate point appears twice in the sum, this can be simplified to

$$h\frac{f(a) + f(b)}{2} + \sum_{i=1}^{N-1} hf(a+hi).$$

Add a method to `FunctionTest` that implements this algorithm on `Math.sin()` with the following signature:

```
public static double defIntegral(double a, double b, int N);
```

Use this method to compute $\int_0^\pi sin(x)$ and $\int_0^{2\pi} sin(x)$. Choose N to be a power of two such that the difference between using $N/2$ and N is less than 10^-8 (ie start at $n = 1024$ and double each time). [3]

Save your code. You will submit it as part of Problem 4.

Problem 3: A Polynomial Class

Write a class `Poly` representing polynomials with integer coefficients. Implement the following methods:

- `Poly(int[] coef)` – Constructor from an array of coefficients c[] where c[n] is the coefficient of x^n.
- `int degree()` - returns the power of the highest non-zero term.
- `String toString()` - returns a string representation of the polynomial (use “x” as the dummy variable and format high-order to low-order powers).

Add addition and multiplication in both static and instance forms:

- `Poly add(Poly a)`
- `Poly mul(Poly a)`
- `static Poly add(Poly a, Poly b)`
- `static Poly mul(Poly a, Poly b)`

Rather than implement substraction, implement a scale method, which multiplies all coefficient by a constant value. Subtraction can them be implemented as p1.add(p2.scale(-1));

[2] Remember the following property of integration: $\int_a^b f(x)dx = \int_a^c f(x)dx + \int_c^b f(x)dx$

[3] In order to find an adequate value of N for a given problem, one must iterate over increasing N and re-computing the integral. In this iteration, a lot of redundent work is performed. If N is doubled each time, it is possible to improve this algorithm by only computing the sum for points not computed on the previous iteration (computed with $N/2$ segments). This sum can be added to the result of the previous iteration to give the correct approxiation for N segments. For further refinements, consult any numerical algorithms text.

- Poly scale(int s)

Comment your class with Javadoc compatable comments and run Javadoc to produce documentation. Hint: Use the `-d` option to have the documentation sent to a subdirectory of your working directory.

Write a driver class and test your implementation.

Download our PolyTest.class driver into your working directory and run it.

Questions:

This is an immutable implementation of polynomials. Unary operators like scale() return new polynomials. What are the advantages and disadvantages of this choice?

What are the advantages and disadvantages of choosing static methods for binary operators vs. instance methods? (In this case we do both).

Submit: The source to Poly.java and your test driver, the output of PolyTest, and your answers to the questions.

Problem 4: Inheritance

We will now use inheritance to generalize the numerical routines from the previous section.

Write a class `RFunc` which will represent the behavior of functions over the real numbers (or in our case, Java `doubles`). Since there is no such thing as a generic function, make `RFunc` an abstract class supporting the method

- `public abstract double evaluate(double x)`

Generalize the bracketRoot and defIntegral methods you wrote earlier, by replacing the calls to Math.sin() with calls to evaluate(), and add them as instance methods to RFunc. (Remember to document RFunc with Javadoc comments).

- `public double bracketRoot(double a, double b, double maxErr)`
- `public double defIntegral(double a, double b, int N)`

Now that we have an abstract function class, we need some real functions. Write subclasses of RFunc, SinFunc and CosFunc, that override the evaluate() method to return Math.sin(x) and Math.cos(x) respectively. Add JavaDoc comments everywhere, as always.

Write a test driver to find the root of cos(x) between 1 and 3. Compute the integral of cos(x) from 0 to $\pi/2$ and 0 to π.

Download and run the FuncTest.class driver in your working directory and submit the results.

Submit: The source to RFunc.java, the output of FuncTest, and your answers to the questions.

Problem 5: Polynomials as Functions

Since polynomials can be considered as functions, modify Poly to be a subclass of RFunc (add an evaluate() method). Modify your driver class to compute the positive root of $x^2 - 3$ and $x^2 - x - 1$. Also compute the integral of $x^2 - 4$ from 0 to 2.

Re-run Javadoc on your function classes and browse the result.

Unlike arbitrary functions, polynomials can be integrated in closed form.

$$\int_a^b f(x)dx = F(b) - F(a)$$

where $F(x)$ is the indefinite integral of $f(x)$. For a polynomial $\sum a_i x^i$ the indefinite integral is $\sum \frac{a_i x^{i+1}}{i+1}$. Use this principle to override the defIntegral method in Poly. Use your test program to again compute the integral of $x^2 - 4$ from 0 to 2.

Download and run the FuncTest2.class driver in your working directory and submit the results.

Submit: The source to Poly.java and the output of Func2Test.

Problem 6: Interfaces

An alternate way to abstract functions is as an interface. We can write an RFuncLib class to be a repository for the bracketRoot and defIntegral methods, and use an interface `Function` to hold the `evaluate` method.

Write a `Function` interface with one method, `evaluate`. Modify SinFunc, CosFunc and Poly to implement Function rather than extend RFunc. Write the RFuncLib class to contain your numerical routines as static methods. You will need to modify them to take an additional argument of type `Function` and use this as the function to evaluate. Test this new implementation.

Download and run the FuncTest3.class driver in your working directory.

Questions:

What are some reasons for choosing the inheritance-based approach rather than interface-based approach and vice-versa?

Submit: The source to RFuncLib.java, Function.java, the output of FuncTest3, and answers to questions.

Problem 7: Class Design

Design a class hierarchy (classes and/or interfaces) to support a program dealing with geographic objects. Support the following classes (at least):

- Countries
- States/Provinces
- Cities
- Boundary Segments
- Rivers

Support the following operations, where applicable, plus any others that seem useful (arguments have been omitted):

- area()
- capital()
- getCities()

- getCountry()
- distance() – between cities
- boundaryLength() – total length of boundary
- neighbors() – objects sharing boundaries
- borderOf() – the countries/states this separates

Write out the class definition, instance vars and method definitions. Don't bother to implement the methods (but make sure you could). Use interfaces and superclasses where appropriate. Supply javadoc comments for all classes, interfaces, and methods.

Note: This problem is deliberately openended. Don't panic!

Submit: Your class and method definitions (in a single text file).

Problem 8: Priority Queue

Priority queues are containers that hold objects that can be compared. That is have an order relation equivalent to $>$. Object are inserted into a priority queue arbitrarily, but are removed in sorted order. That is, the largest (or smallest) element in the queue is removed and returned. The basic PriorityQueue interface is

```
interface PriorityQueue{
    /**
     * Add an Object to the queue
     */
    public void insert(Comparable a);
    /**
     * Removes and returns the maximum object from the queue
     */
    public Comparable removeMax();
    /**
     * Returns true iff queue is empty
     *
     */
    public boolean empty();
    /**
     * Returns the number of objects in the queue
     *
     */
     public int length();
}
```

These queues can only hold Object classes which implement the Comparable interface. (Note: This interface is defined in package java.util so you needn't redefine it.)

```
interface Comparable{
    /*
     * Return -1,0,or 1 depending on whether 'a' in less-than, equal to,
     *  or greater than the implicit arg (ie 'this') in the desired
     *  ordering.
     */
    public int compareTo(Comparable a);
}
```

Write a class PQueue that implements PriorityQueue. [Don't worry about efficiency. You can use a linked list, doubly-linked list, array, or Vector as the underlying data structure. It is easiest to sort the objects as they are inserted. Make sure the structure can grow to arbitrary size, yet properly shrinks when items are removed.]

The Java String and Integer classes implement Comparable. Test your class by inserting a handful of Strings and removing them, verifying they come out in the right order. Test that it still work when you interleave inserts and removes.

Modify your Poly class to implement Comparable. The ordering for polynomials: compare degrees first, if degrees are equal, then compare leading coefficients, if these are equal, compare on lower order terms.

Questions:

What happens if you insert some Strings then some Integers (remember: this is the class wrapper for int, not the int type). What happens if you try to removeMax()?

This reveals a problem with this type of abstraction. Is there any good solution?

The java.utils package has a class TreeMap which implements similar functionality to PriorityQueue. It can use the Comparable interface to sort, but it also allows the specification of a Comparator object in the constructor. A Comparator is a class with a binary compare() function. The significant difference is that the Comparator is attached to the TreeMap rather than the elements themselves (as is the case with the compareTo method of Comparable). Does this solve the problem encountered above?

Submit: PQueue.java and answers to questions.